For a New Real-Time Methodology

Thierry Gautier, Paul Le Guernic, Olivier Maffeis

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

For a New Real-Time Methodology

Thierry GAUTIER Paul LE GUERNIC Olivier MAFFEÏS

N° 2364

October 17, 1994

PROGRAMME 2

ISSN 0249-6399

For a New Real-Time Methodology

Thierry GAUTIER* Paul LE GUERNIC** Olivier MAFFEÏS***

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet EP-ATR

Rapport de recherche n°2364 — October 17, 1994 — 51 pages

Abstract: This paper motivates the use of a synchronous methodology to program, to verify and to implement real-time applications. The main features of the synchronous language SIGNAL are presented and some methodological principles are proposed. SIGNAL programming is illustrated through the development of a realistic application, a complex digital watch. Then, an overview of the software environment associated with SIGNAL is presented. This environment encompasses formal verification as well as implementation tools to infer reliable real-time implementations on various architectures including distributed ones. All these tools are acting on a unique but polymorphic abstract program representation, namely Synchronous-Flow Dependence Graphs, which provides the SIGNAL software design environment with a great homogeneity.

Key-words: reactive & real-time systems, synchronous approach of time, the SIGNAL language, specification/programming methodology, software design environment, synchronous-flow dependence graphs, computer-aided formal verification & implementation.

(Résumé : tsvp)

*IRISA/INRIA – Campus de Beaulieu – 35042 Rennes Cedex – France – e-mail: Thierry.Gautier@irisa.fr
**IRISA/INRIA – Campus de Beaulieu – 35042 Rennes Cedex – France – e-mail: Paul.LeGuernic@irisa.fr
***GMD I5 - SKS – Schloß Birlinghoven – 53721 Sankt Augustin – Germany – e-mail: Olivier.Maffeis@gmd.de
***Olivier Maffeïs is supported by an ERCIM post-doctoral fellowship.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Pour une nouvelle méthodologie temps-réel

Résumé : Cet article prône l'utilisation d'une méthodologie synchrone pour programmer, vérifier et implémenter les applications temps-réel. Les principaux traits du langage synchrone SIGNAL sont présentés et quelques principes méthodologiques sont proposés. La programmation en SIGNAL est illustrée à travers le développement d'une montre digitale complexe. Une vue générale de l'atelier de conception de SIGNAL est ensuite présentée. Cet atelier comporte aussi bien des outils de vérification formelle que des outils d'implémentation. Il permet l'inférence de mises en œuvre sûres et efficaces sur diverses architectures, dont les architectures parallèles à mémoire distribuée. Tous ces outils opèrent sur une unique bien que polymorphe représentation abstraite des programmes, les graphes de dépendances de flots synchronisés, ce qui procure une grande homogénéité à cet atelier de conception.

Mots-clé : systèmes réactifs & temps-réel, approche synchrone du temps, langage SIGNAL, méthodologie de spécification/programmation, atelier de conception, graphes de dépendances de flots synchronisés, vérification formelle et implémentation assistées.

1 Introduction

Real-time computing encompasses applications from missile control to ballistic defense systems, from vehicle and train control to space shuttle and avionics, from workshop automation to nuclear power plant. Whereas logistics, economics and human stakes are critical in these applications, the design of real-time systems are mainly performed using craft methods or design environments which provide the designer with limited help often reduced to the organization and the presentation of his thoughts. In this paper, we advocate a new real-time methodology and present the software design environment we developed. This methodology intends to include all the development steps of a real-time system, from its early design up to its implementation on a particular architecture, each step being supported by formal verifications.

The presentation of this new methodology for the design of real-time systems is divided in five parts. Firstly, we set up the context of this methodology by drawing the characteristic features and the essential requirements of real-time applications. Secondly, we explain the fundamental basis of our real-time methodology: the synchronous approach of time. Thirdly, we present the specification language SIGNAL on which our methodology is based. The specification methodology we advocate is then illustrated by programming a complex real-time application: a digital watch including a stopwatch and an alarm. Finally, we draw the main features of the SIGNAL software design environment and the tools it includes to verify and implement the SIGNAL programs on various architectures.

1.1 Real-Time Features and Requirements

(i) Interactivity

A real-time system, as a *timely reactive system* [17, 29], monitors and tries to control its environment by maintaining an on-going interaction with it. The environment of a reactive system may include physical devices that require extremely frequent interactions (the radar in missile defense systems). One the other hand, the environment may embrace relative slow performance systems like high-level decision centers (human or expert systems) to supervise complex activities (the choice of air corridors in air-traffic control systems) or exceptional ones (financial crashes). Consequently to this heterogeneity of their environment, reactive systems have to cope with on-going interactions taking place on *various time-scales*.

(ii) Parallelism & Portability

A real-time system is *essentially a parallel system* since it interacts with a heterogeneous environment, each part of this environment having its own evolution. Real-time systems go from small applications (embedded systems) which require drastic resource consumption (time, memory) up to wide ones (air-traffic control, telemetry control systems) in which geographical constraints may enforce an implementation through parallel processes —complex applications are often composed of several cooperative smaller real-time systems, some of them being embedded.

Besides the intrinsic parallelism of real-time systems, they are generally designed to ensure a long-lived service. For instance, the life of a nuclear power plant may be 10 or 20 years. Therefore, the specification of the real-time systems must be portable and evolvable to be able to cope with the modifications of the functionality and the evolution of the operational techniques along such periods.

Thus, durability, speed issues and interactions on various space-scales imply that the specification of a real-time system should be *architecture independent* —see also [35]— and enable *parallel executions*.

(iv) Temporal Constraints & Predictability

In the relationship with its environment, the real-time system is fully responsible for the synchronization; if the synchronization with its environment does not correctly occur, it would lead to distortion phenomena. To synchronize properly with an environment in which time is an essential component of its behavior, a real-time system has to handle time and *timing constraints*. We distinguish two quantitative ways of expressing timing constraints:

- *response-time constraints (latency)*, for which the durations, between an event and the induced reaction of the system, must be considered. These constraints highly depend on the application area (tens milliseconds for an aircraft, a few seconds for a reservation system, ...); response-time constraints are expressed over a chronometrical representation of time;
- *rate constraints*, for which we have to consider the number of events that can be processed by the system during one time unit; rate constraints are expressed over a chronological representation of time.

A real-time system behaves in real-time if its implementation verifies these timing constraints. Therefore, to verify the real-time feature, *the temporal behavior of the system and its implementation must be predictable in time*.

(iii) Dependability

The simple list of some application areas is sufficient to convince oneself of the necessity to give a special care to dependability: nuclear power plants, ballistic defense systems, spacecraft systems, ...

To let dependability rely *exclusively* on the designer/programmer's skills does not provide enough safety guarantees. Moreover, this way of designing is intractable and finally inefficient for large applications.

One should provide formal verification techniques dealing with properties of the specification (deadlock freedom, mutual exclusion of states, responsiveness, ...). Moreover, one should supply tools which enable the inference of highly-efficient, time-predictable and even fault-tolerant implementations while preserving the equivalence with the specification of the system.

1.2 Real-Time Systems

(i) Definition

A real-time system is a system which maintains an on-going interaction with an environment in which time is an intrinsic and essential component of its behavior; a real-time system is a timely reactive system. By means of this interaction, the real-time system monitors its environment, synthesizes diagnostics and may emit commands to direct it. For instance, a real-time system which controls a nuclear power plant must prevent any overheating of the core. Consequently, it has to estimate as accurately as possible the current state of the core with respect to the inputs provided from sensors. According to the predictable evolutions of the core from its current state, the system may alarm the operators if some overheating is foreseeable, and it may produce adjustment commands to actuators.

As time is an intrinsic and essential component of the behavior of the environment, the reaction of the system is subjected to some *temporal limit*[1] beyond which the meaning of the information is changed for the environment. As an extreme example, after its takeoff, the reservation of a place in the plane becomes useless. Roughly, although the relative evolution of the real-time system and its environment is fully asynchronous, the reaction of a real-time system must seem instantaneous

[1] This temporal limit may be very strict (i.e., the cost of a failure is very high) and the system is called a *hard real-time system*. On the other hand, this limit may be less strict and the system is a *soft real-time system*. Complex applications are often an intricate composition of both kinds of systems.

according to the time granularity of the environment changing state. This point of view should be of adjustable granularity according to the level of observation.

(ii) Structure of a Reactive System

Let us describe more precisely the generic structure of the systems we consider. We call *co-system* a reactive system whose structure is depicted in the grey box of Fig. 1.

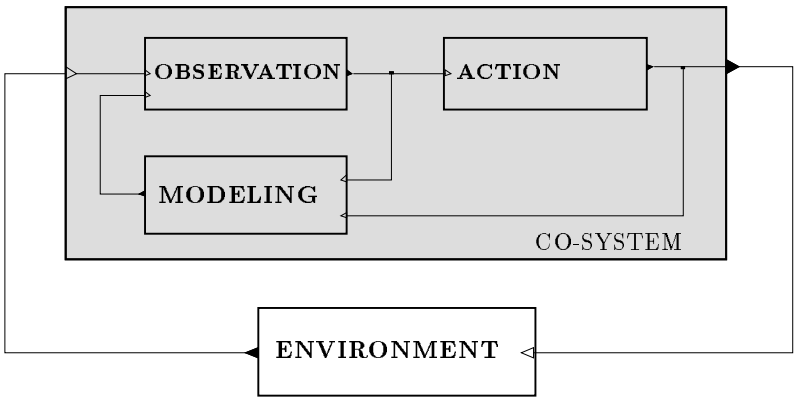


Fig. 1: Generic Structure of a Real-Time System

The co-system receives data from its environment (represented by the left link in Fig. 1) and sends data to it right link). Depending on the application, the data communicated between the co-system and its environment are quantitative (for example, the speed and the position of a lift) or qualitative (for instance, the state opened/closed of its doors).

Regardless of the monolithic depiction of the environment, it may have a very complex structure *just like a concurrent co-system* except that its own internal dynamic is visible only through its interface. In this representation, we neither give greater importance to the co-system (transformational approach), nor to the environment (pure reactive approach [17]); the relative influence of the co-system and its environment is very symmetrical: the evolution of the environment influences the co-system through some sensors and the co-system affects its environment through

commands to actuators.

As considered in the control automation community, a *co-system* is generally composed of the following functional components:

- an **OBSERVATION** module, which records the observed evolutions of the environment; these evolutions may concern states of the monitored devices or commands from the high-level decision centers;
- an **ACTION** module, which synthesizes diagnostics and produces commands for different parts of the environment;
- a **MODELING** module, which is in charge of capturing the environment behavior. According to the current observation and the previous modeling, some distance between the predictable behavior and the current state of the environment may be estimated. This estimation may induce an adjustment of the modeling and influence the commands emitted to direct the environment.

According to the performed interaction, a taxonomy of co-systems, in increasing order of generality, may be:

- *functions:* their behavior is, at each occurrence of data on their inputs, the calculation of a function (in the mathematical sense) from its inputs to its outputs. Examples of devices implementing functions are bells, ALU, FFT chips, etc.
- *deterministic automata:* their behavior is, at each instant, the computation of a function from its inputs and a current state to its outputs. With respect to the active instants of these automata, we can distinguish *purely reactive automata,* for which these instants are defined by the occurrences of inputs, and *generative (i.e., output-sensitive) automata,* for which they are defined by the occurrences of outputs. An example of generative automaton is the quartz crystal in a digital watch. By enabling the specification of both reactive and generative automata, mixed passive and active systems can be described, thereby a complete symmetry between a co-system and its environment is considered;
- *relations* (non-deterministic automata) which set, at each instant, some relation (in the mathematical sense) among their inputs, their current state and their outputs. For instance, random functions, noisy transmission systems, or partial specifications belong to this class of systems.

If one does not want to implement relations, at least, it may be necessary to describe them to simulate their behavior, and to obtain, by refinement, a deterministic system. The purpose of our real-time design methodology is to describe all those types of reactive systems although only the first two classes of reactive systems are considered for a real-time implementation.

1.3 The problematics of Time

Basically, the specific characteristic of the design of a real-time system is the necessity to consider time as an intrinsic element. Starting from a short description of the physical time, we present two different approaches of time in parallel programming languages: the asynchronous approach and the synchronous one.

1.3.1 Physical Time

(i) Measurability

Time may occur as a measurable physical entity in the behavior of the processes that a real-time system has to control. For instance, time may be expressed through equations of dynamics which allow a quantitative and continuous description of the behavior of the object as a function of time. In this approach, time is considered under its *chronometric* aspect: the duration has a significant role in actions (the application of a non timed force for moving a lift would smash it on the top or on the floor).

Another approach to temporal phenomena is to focus on qualitative properties. For example, the lift should not go up or down when the doors are opened (independently of its speed). Considering qualitative properties leads us to handle the *chronology* and the *simultaneity* of events. More precisely, we have to abstract a continuous phenomenon by a denumerable set of events (or states). These events could be, for instance, "the doors become closed", or "the button of the second floor is pressed". The interactions between the system and its environment will be described over this set of events.

(ii) Non Isotropy

A particular property linked to the concept of time is the principle of *uncertainty* stated initially by W. Heisenberg: due to the non null duration of information transfers, it is not possible to know the current state of a system. This uncertainty significantly

disrupts the processes of knowledge acquisition (it is particularly manifest in astronomy; concerning real-time systems, the problem is increased by the variations of quartz frequencies).

Another particular property of time as a physical entity is its *irreversibility*, which in chemistry is expressed by Carnot's principle. The increase of entropy in a system leads to consider only *causal* models in real-time systems specifications: it is not possible to compute the current state of the system according to its future; at best, it is only possible to estimate the future of a system with respect to its estimated (uncertainty principle) current state.

Thus, time is a measurable and non isotropic physical entity; an extended presentation of the different aspects of time is given in [37].

1.3.2 The Asynchronous Approach

According to the requirements of real-time systems, their behavior must be specified in a parallel and portable language. The most common time model for concurrent programming is asynchrony. As R. Milner wrote in the introduction to CCS [32]:

> *"We aim to describe a concurrent system fully enough to determine exactly what behavior will be seen or experienced by an external observer.*
> *...*
> *When we compose two agents it is the synchronization of their mutual communications which determines the composite; we treat their independent actions as occurring in an arbitrary order but not simultaneously"*

Thus, an asynchronous language relies on the observed interleaving of perceptible events which may happen during execution. We consider as *asynchronous* any system of representation of events (some atomic abstraction such as rendez-vous) that does not allow to perceive simultaneous events. Examples of asynchronous real-time languages are CSP [18, 33], ADA [36, 2] and Real-Time Concurrent C [13].

As the asynchronous approach relies on the observed interleaving of perceptible events during execution, this approach is implementation-oriented; it forces to consider more or less abstractly the target architecture at early specification stage of the applications. Therefore, the portability of the specifications expressed in an asynchronous parallel language is, to some extent, limited.

The second drawback of the asynchronous approach concerns the safety which may result from formal verification techniques. Since the asynchronous approach requires to introduce implementation considerations at the specification stage, the

semantics of the implementation interferes with the proper semantics of the application, thereby some of its intrisinc properties may be masked. This possible loss of information may affect the demonstration capability. Besides this loss of information, the difficulty to prove properties over asynchronous specifications is even greater with the non-determinism: non-determinism increases the complexity of proof by increasing the number of sequences of events that may occur at run-time.

1.3.3 The Synchronous Approach

A first approach to synchrony has been investigated by Milner in [31]. In this subsection, we present another approach which, in contrast with Milner's one, can be seen as a coarse-grain approach to synchrony. Introductory papers to this approach are [9, 4].

(i) Principle

In the asynchronous approach, executing an operator consumes some time; time is considered at a fine-grain physical level. In the synchronous approach, *time is abstracted from the computations and the communications* (i.e., computing and communicating take no time): *time is transfered on the data level and related to their state changes*; time is considered at a coarse-grain logical level. Hence, while the asynchronous approach consists in focusing on the computational aspects in the description of behaviors, the synchronous approach adds a level of abstraction by focusing initially at a logical level. Therefore, the description of behaviors is less dependent from the implementation issues; the synchronous approach induces application-oriented specifications, thereby it improves their architecture-independence and increases their portability.

One possible way to characterize the notion of *state change* on the data level is to associate this approach to synchrony with the dataflow approach and its single assignment principle. Then, the time elapsing at the data level is simply denoted by the precedence of successive values on the same signal (flow of data): two values on a signal belong to two different instants of execution, the succession of the values denoting a sequence of execution instants.

With this point of view, the events are observed *chronologically* taking into account their possible *simultaneity*. In this approach, time is a discrete domain; it may be generated by purely temporal events (for instance, the seconds given by a physical clock), or by events resulting from the evolution of an environment device in non temporal measures (for instance, the passing of a lift to the floors).

(ii) Programming Style

As the synchronous approach leads to abstract execution and communication durations, some actions become assigned to the *same logical instant of execution*. Therefore, whereas with an asynchronous approach behaviors are specified as a "continuous" succession of computation steps and waiting steps for synchronization, the synchronous approach induces a discretization of this operational behavior, each fragment of behavior resulting from this discretization being executed in "zero time". Consequently, an application is specified as sequences of instantaneous macroscopic reactions to stimuli: *an application is specified with the synchronous approach as sequences of reflexes; the synchronous approach of time is the basis of the reflex programming style*.

According to this latter remark, the different existing synchronous languages can be divided into two groups: the imperative ones (STATECHARTS [16], ARGOS [30] and ESTEREL [10]) focusing on the sequencing, and the declarative ones (LUSTRE [15] and SIGNAL [25]) focusing on the reflexes.

(iii) Synchronous Paradigm

A paradigm for the synchronous approach is the definition of a time counter. Let us consider some external device that delivers a tick every second. In order to produce the ticks for the minutes, we build a counter modulo 60. In an asynchronous approach, this behavior will be specified as an iteration over the sequence: wait for the second tick then increment the counter or, if it reaches 60, reset it and emit a minute tick. The property that a minute occurs every sixty seconds is trapped in the implementation-oriented specification: no proof or validation system may consider this property. By contrast, in the synchronous approach, the same behavior will be specified as two exclusive reflexes: (a) increment the second counter or (b) reset the counter and emit a minute tick. The event *minute* is defined as occurring in the same reflex as the reset of the second counter: the event *minute* occurs simultaneously with the sixtieth event *second,* and this can be used in proofs.

(iv) Resource Bounds

An implementation of a system is said real-time if the latency (response time to some stimuli) is bounded by the latency of the state change of the environment. A system will be able to accept a real-time implementation only if the resources it requires can be bounded. Over the time resource, this property imposes that we must be able to ensure that the number of operations in each different reflex is bounded. Over

the memory resource, we have to assert that only a bounded memory is requested. These verifications are eased thanks to the discretization of the behavior occurring at the specification level since they are verified by restricting the study to properties of the reflexes and not of their composition.

Thus, with the synchronous approach, it is possible to specify, to prove, and to simulate the functional behavior of real-time systems. When and only when the specification has been certified at the functional level, we can consider its implementation on a particular architecture and verify if it runs in real-time.

(v) Handling Physical Constraints

With the synchronous approach, a real-time system is specified as sequences of reflexes to stimuli. This approach comes down to initially remove the physical durations from the specification; these physical durations are only inserted when they are required, at the implementation stage. The abstraction into reflexes is still very useful at this stage since it is over the reflexes to stimuli that the real-time feature of an implementation is verified.

Over an implementation on a target architecture, this verification of the real-time feature can be analytic by estimating at compile-time the worst response time (a bound of the response time exists thanks to (iv)). If such an analytic verification is not possible since some feature of the target architecture does not enable bounded response time, some watchdog events must be added to the initial program to assert the timing constraints.

In the sequel of this paper, we emphasize the specification of real-time systems using SIGNAL: in section 2, we present the SIGNAL language, a synchronous and dataflow language; in section 3, the methodology of specification is presented and illustrated by programming a complex digital watch. Finally, in the later sections, we present the software design environment which has been developed to deal with SIGNAL specifications; the significant property of this environment is its homogeneity which results from the unique abstract program representation on which all the verification tools as well as the implementation techniques act.

2 SIGNAL: a Synchronous Dataflow Language

The synchronous approach is the base of the class of *synchronous languages* which includes in particular the STATECHARTS [16], ARGOS [30], ESTEREL [10], LUSTRE [15],

and SIGNAL —see [14] for an overview of the three latter ones. In this section, we introduce the SIGNAL language, a synchronous and dataflow language. The reader interested in introductory papers on the language may look at [6] and [25]; a more detailed presentation is given in [7] —and for the historians, the first published paper on SIGNAL is [24].

As SIGNAL is a dataflow language, it handles (possibly infinite) sequences of data with an implicit dating: the *signals.* At a given instant, a signal may have the status *present* (it holds a value) or the status *absent* (denoted by $\perp$). If `X` is a signal, we denote by $(\mathtt{x}_t)_{t\geq 1}$ the sequence of its values when it is present. Signals that are always present simultaneously are said to have the same *clock* or to be synchronous; clocks can be considered as equivalence classes of synchronous signals. The operators of SIGNAL are intended to relate clocks as well as values of the various signals involved in a given system. We term a system of such relations *process;* processes may be used as modules and further combined as indicated later.

2.1 Elementary Processes

- `Y := f(X1,...,XN)` denotes a function (or even a relation) on values extended to act on signals (sequences of values). Let `f` a symbol which denotes a n-ary function acting on signals (arithmetic or boolean operations, user-defined functions) and f the corresponding function acting on values, the above SIGNAL-expression defines the elementary process which satisfies:

  $$\forall t \geq 1 \quad \mathtt{y}_t = f(\mathtt{x1}_t, \ldots, \mathtt{xn}_t)$$

  A byproduct of these operators is that all the referred signals must be synchronous, i.e. they must have the same clock.

- `Y := X $1` denotes a *delay* operation; this expression represents an elementary process such that

  $$\forall t > 1 \quad \mathtt{y}_t = \mathtt{x}_{t-1}, \quad \mathtt{y}_1 = y0$$

  where $y0$ is an initial and constant value associated with the declaration of the signal `Y`. Just like the previous one, this operator forces the referred signals to have the same clock.

- `Y := X when B` denotes a sampling of a signal through a boolean condition. The signal `Y` takes the value of `X` if both `X` and the boolean signal `B` are present,

and B is carrying the value *true*; otherwise, Y is absent. A possible sequence of values over X, B and Y may be:

X :	1	2	⊥	3	4	⊥	5	6	9	...
B :	*t*	*f*	*t*	⊥	*t*	*f*	⊥	*f*	*t*	...
Y :	1	⊥	⊥	⊥	4	⊥	⊥	⊥	9	...

(`Y := X when B`)

- `Y := U default V` denotes the deterministic merging of signals. The output Y is equal to U when U is present, otherwise it is equal to V (and then absent if both U and V are absent); this priority given to U makes this merge deterministic. A possible sequence of values over X, B and Y may be:

U :	1	2	⊥	3	4	⊥	5	⊥	9	...
V :	⊥	⊥	3	4	10	8	9	2	⊥	...
Y :	1	2	3	3	4	8	5	2	9	...

(`Y := U default V`)

The above operators are the only ones acting on signals which belong to the kernel of SIGNAL. In addition to these operators, we will use frequently the three following expressions which are directly derived from the kernel of SIGNAL:

- `C := event X` delivers an event-signal C (of type `event`) which is a signal occurring only with the value *true*. C is synchronous to the signal X: it is the clock of X. A possible sequence of values over X and C may be[2]:

X :	1	2	⊥	3	⊥	⊥	5	6	9	...
C :	*t*	*t*	⊥	*t*	⊥	⊥	*t*	*t*	*t*	...

(`C := event X`)

- `C:= when B` delivers an event-signal C when the signal B is equal to *true*; `C:= when B` is the shortened form of `C:= B when B`.

- `synchro { U, V }` is a process without output that constrains its inputs to be synchronous.

[2] `C:= event X` is rewritten as `C:= (X=X)` in the kernel of SIGNAL.

2.2 Process Composition

The parallel composition (| P | Q |) of two processes P and Q is quite similar to the union of sets of equations: an input signal X of P has the values defined in Q if X is an output signal of Q and vice versa. A signal cannot be defined in more than one process; the input signals of (| P | Q |) are those of P and Q which are not defined in neither of them; the output signals of (| P | Q |) are those of either P or Q. This composition operator is commutative and associative.

2.3 Data Types

The values held by signals may belong to the following types:

- the control types (`event` and `logical`) used in relations and functions;
- the numerical types (`integer, real` and `complex`) used in functions;
- multi-dimensional arrays of any type; the elements of an array have the same type; moreover, they are synchronous.

The next version of the SIGNAL language, currently under development, will include several other types like strings, records, tuples, etc.

2.4 A Simple Example

The following SIGNAL process emits a tick every minute —event-signal TM— and the number of seconds —signal S— elapsed since the last minute tick. The only input of this process, called SECOND, is an event-signal TS delivered by its environment (we assume that TS is received every second):

```
(| synchro { S, TS }
 | ZS := S $1
 | S := ( 0 when TM )default( ZS+1 )
 | TM := TS when( ZS=59 )
 |)
```

- `synchro { S, TS }` ; the occurrences of the integer signal S and that of the event-signal TS are synchronous: S counts the occurrences of TS;
- `ZS := S $1` ; the integer signal ZS carries, at each one of its occurrences, the previous value of S; both signals are implicitly synchronous;

- `S := ( 0 when TM )default( ZS+1 )` ; the signal `S` counts the number of ticks of `TS` modulo 60: it is reset every minute (occurrences of the event-signal `TM`); otherwise, it takes its previous value (carried by `ZS`) incremented by 1;
- `TM := TS when( ZS=59 )` ; the event-signal `TM` and the occurrences of `ZS` with `59` are synchronous: a tick for the minute is produced every 60 seconds.

A run of the SIGNAL process `SECOND` produces the following sequences of values on its signals:

`TS`	:	$\cdots$	t	t	t	t	t	t	t	t	t	t	$\cdots$
`TM`	:	$\cdots$	$\perp$	$\perp$	$\perp$	$\perp$	t	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\cdots$
`ZS`	:	$\cdots$	55	56	57	58	59	0	1	2	3	4	$\cdots$
`S`	:	$\cdots$	56	57	58	59	0	1	2	3	4	5	$\cdots$

2.5 Modularity

Any SIGNAL process may be "encapsulated" in a process frame which allows one to abstract any process to an interface; the process can be used afterwards as a black box through its interface. The interface of a process describes its input-output signals and its parameters (dimensions of arrays, initializations, ...). The process frame enables the definition of sub-processes; the sub-processes which are just specified by an interface without an internal behavior are considered as external (they may be separately compiled processes or physical components).

Example

Let us make the previous simple example a more generic one through the process `WHEEL` depicted in Fig. 2; this figure is a screen-dump of the graphic interface which is included in the SIGNAL software design environment.

This process specifies a modulo counter of the occurrences of the input signal `TICK`; this process supports an additional external reset by means of the signal `RESTART` (both `TICK` and `RESTART` are `event` type signals or, in short, event-signals). The output signal `V` holds the number of occurrences of `TICK` until the last reset; the output signal `TOP` occurs when the value of the modulo counter has reached its limit specified by the (constant) parameter `MOD`.
The body of the process `WHEEL` (Fig. 2) is made up of two components:

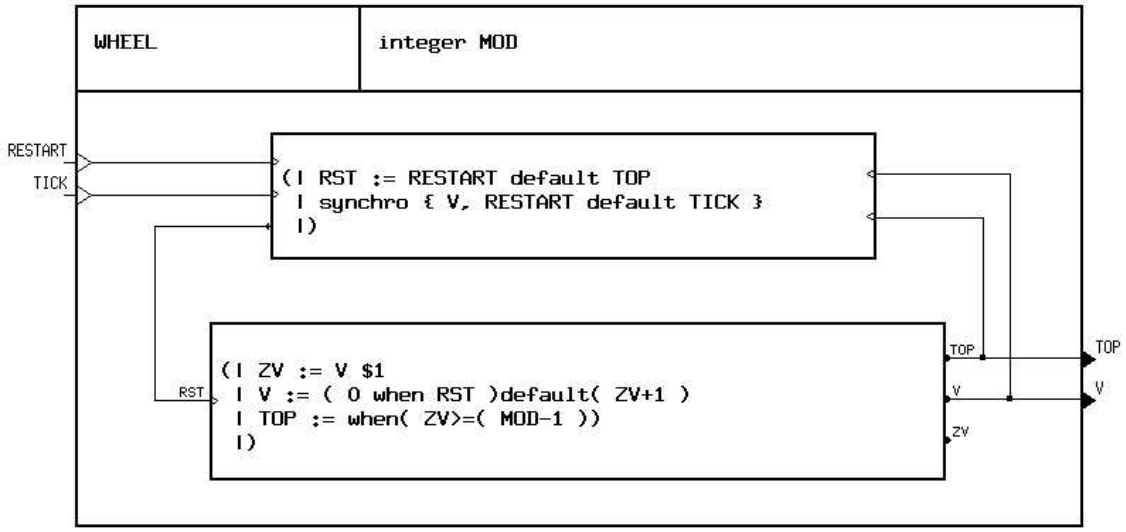


Fig. 2: The Process WHEEL

- *the bottom box* specifies the same behavior as in the previous example except that (a) the limit is stated by the parameter MOD (a constant value) and (b) the reset of the counter is not restricted to the occurrence of TOP as specified in the top box.
- *the top box* specifies that the counter is reset —occurrences of RST— when an external reset is requested —occurrences of RESTART— or when the counter has reached its limit —occurrences of TOP. The second line of the top box extends the synchronization in the previous example to cope with the external resets.

The process WHEEL will be used in the next section to specify in SIGNAL the counting heart of the stopwatch.

The overall language includes several other operators to ease programming, all these operators are derived from the kernel of SIGNAL for which a mathematical semantics has been defined [7]. The description of the overall language is out of the scope of this paper —see [11] for a complete information.

3 Describing a Digital Watch in SIGNAL

Through the description of this example, we would like to illustrate some methodological principles to design complex real-time systems in SIGNAL; these principles have been motivated by safety properties as well as software engineering requirements like portability, maintainability, modularity and reusability —see [19] for a survey of the design methodologies for real-time systems. Besides drawing up a methodology for programming in SIGNAL, we would like to present the ability of the language to handle internally or externally generated interruptions, exceptions, data-dependent (down)sampling, mixed passive/active communications with the external world, etc., which are control structures commonly used in real-time programming. To specify the digital watch, we introduce very high-level processes. These processes have been designed to be as generic as possible; the modularity of SIGNAL enables their re-use in completely different environments.

3.1 Brief Description of the Application

We chose to describe a digital watch as a typical real-time application. The digital watch we specified is widely inspired from the programming benchmark used in [16] and [8]. This digital watch (Fig. 3) has three modes:

- a *watch* mode, that shows the time, the date and the day of the week;
- a *stopwatch* mode with a 1/100 sec precision;
- an *alarm* mode, in which the alarm time can be set to the minute.

The watch and the alarm modes are themselves split in two sub-modes, the regular and the setting modes. This digital watch supports several options: a time-display in AM/PM or in 24h formats, beeps on the occurrence of the hours, etc.
The digital watch has five input event-signals:

- the system time (`QTZ`);
- the upper-left button (`UL`) used to switch from regular to setting modes;
- the lower-left button (`LL`) used to circularly switch the current mode from *watch* to *stopwatch* and *alarm;*
- the upper-right button (`UR`) and the lower-right button (`LR`) have effects which are specific to the current mode.

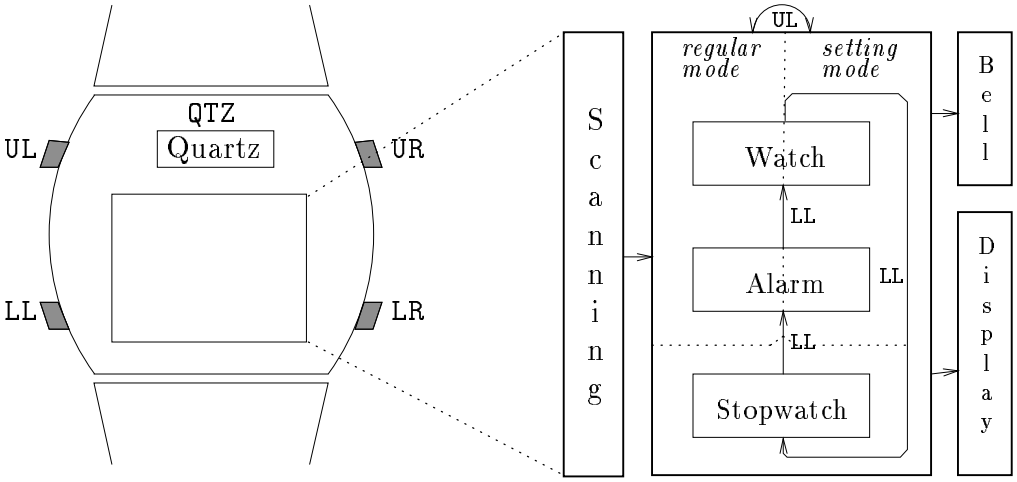


Fig. 3: The digital watch

The design of the digital watch follows roughly a *top-down methodology*. It is composed of two main steps of refinement. Firstly, we draw up three methodological principles from which we decompose the design of the digital watch into simple and homogeneous processes. Secondly, we define two principles related to style of programming in SIGNAL. Finally, we simulate the specified digital watch by programming in SIGNAL its simulation environment.

3.2 Getting Portability

As a real-time system, a digital watch is a time-critical application which maintains an on-going relationship with its environment. One the one hand, as real-time systems are closely related to their environment, their behaviors are deeply affected by the environment features, thereby getting portability in their design may be a very intricate task. On the other hand, as these systems perform usually a long-lived service, they will have to cope with several technological advances: their design *must* be highly portable. Therefore, the first principle of our design methodology concerns portability.

1st Design Principle: **Isolate the technology-related components**

For the design of the digital watch, this principle induces the decomposition achieved in Fig. 4. In this figure, one component has been associated with each kind of interface device. Consequently, if the capabilities of any of these devices are upgraded, the program rewriting can be restricted to the associated component. The ability of the graphic interface to abstract text has been used to mask all the parameters of the processes; these parameters which add generality to the design are abstracted as a `#`. Note that the two inputs `HF` and `LF` of the process `DIGITAL_WATCH` stand for the frequencies at which the events on the right and the left buttons are respectively considered: the left buttons, which produce the mode changes, are relatively less frequently considered since they require more complex reactions. In return, the events on the left buttons have priority to those on the right ones.

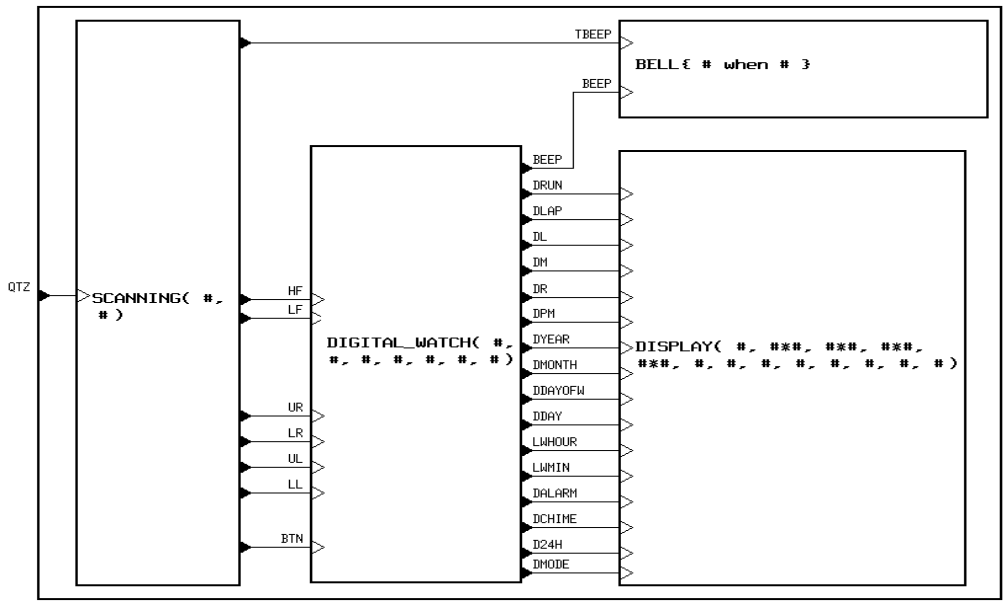


Fig. 4: The `DIGITAL_WATCH` embedded in its Environment

Besides its proper use to cope with the technological advance, this first principle can be used to simulate SIGNAL programs: it is sufficient to replace the interface components by processes which simulate an interface to these physical devices in order to simulate the whole program. This use of the first principle is presented in sub-section 3.7 where we describe a simulation environment for the digital watch.

These first two uses of the first principle were occurring at the interface level; this principle may be used as well in the specification of an application to denote implementation requirements. For instance, this principle may be used to denote processes which will be embedded. It can also be used to describe components which will be implemented in hardware; the SIGNAL environment has achieved a first step towards co-design by producing VHDL code to implement SIGNAL processes [3].

3.3 Getting Maintainability

While portability refers to the facility to cope with changes of the execution environment of an application, maintainability refers to the function that the application is intended to perform. Trying to confine the consequences of an evolution of the requirements leads us to introduce a second methodological principle.

> *2nd Design Principle*: **Decompose a process into functionally-coherent components.**

With respect to this principle, the design of the process `DIGITAL_WATCH` (Fig. 4) is divided in five components (Fig. 5): `WATCH`, `ALARM` and `STOPWATCH` implement the three modes the digital watch supports; these components are coordinated by a fourth one called `MAIN_CONTROL`; the signals produced by these components are multiplexed through `MUXSIG` to get one signal for each interface actuator. Note that the two components `ALARM` and `WATCH` have been gathered together since they share the same two sub-modes, the regular and the setting modes.
In this figure, we have emphasized (by putting a grey halo around them) the communication links involved in the control of the current mode of the watch. One the one hand, the main control unit —process `MAIN_CONTROL`— coordinates the activation of the modules `STOPWATCH`, `WATCH` and `ALARM` by providing them with a boolean signal (respectively `SMODE`, `WMODE` and `AMODE`) indicating their state at the clock `HF`. On the other hand, each module provides the main control unit with an event-signal (respectively `SOFF`, `WOFF` and `AOFF`) when the mode they represent is quit. These latter signals make visible at this level of description the chaining among the three modes of the digital watch: switching off the stopwatch —signal `SOFF` output from the process `STOPWATCH`— is interpreted as switching on the alarm —signal `AON` input to the process `ALARM_WATCH`—, `WOFF` is interpreted as `SON`, `AOFF` is interpreted as `WON`.

Due to length reasons, the sequel of this section focuses only on the design of the stopwatch, the presentation of the other modules is omitted. We end this section by programming in SIGNAL the simulation environment of the digital watch.

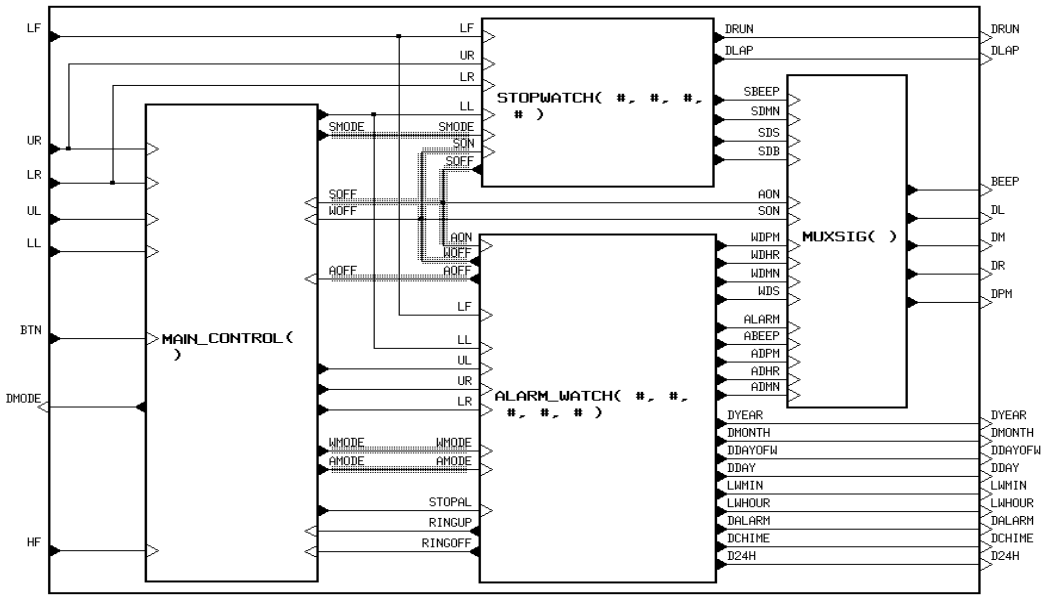


Fig. 5: The Process `DIGITAL_WATCH`

3.4 Getting Modularity

To increase the modularity of the design, a partition resulting from the first two principles may be refined using the third one.

> *3rd Design Principle*: **Split a process into computation and control parts**

According to the first three design principles, the process `STOPWATCH` is made up of four components as depicted in Fig. 6.
These four components are:

- `SCONTROL` (described in the sub-section 3.5), a control component which interprets respectively the buttons `LR`, `UR` and `LL` in:
  - `RUN`: the stopwatch starts and stops counting;
  - `LAP`: the display is frozen/unfrozen if the stopwatch was counting; otherwise, the stopwatch is reset (signal `RESET`);

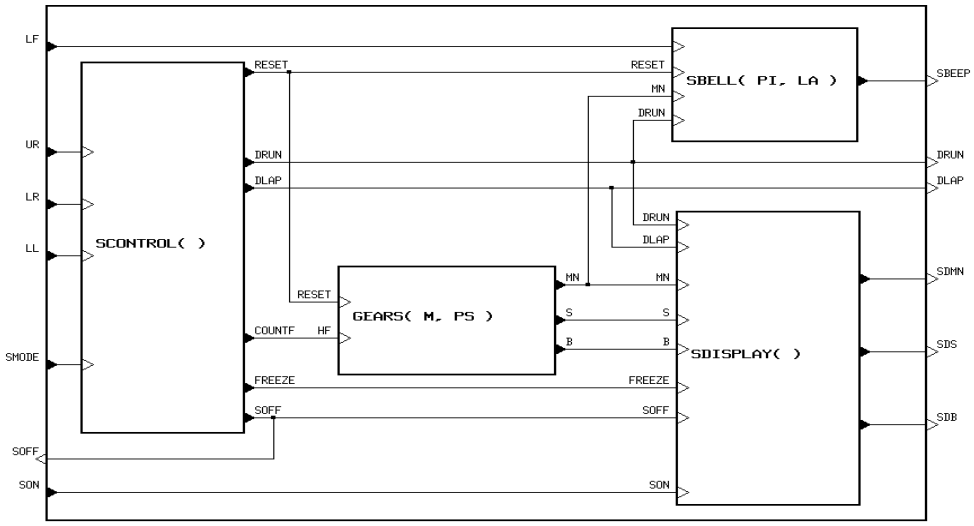


Fig. 6: The Process STOPWATCH

– SOFF: the stopwatch mode is quit.

- GEARS (described in sub-section 3.6), a functional component which constitutes the counting heart of the stopwatch. This counting process is relative to the clock HF and is set to zero on RESET;
- SBELL (not described in this paper), an interface component which synthesizes beep commands with respect to a low frequency clock LF. The bell is rung on RUN, RESET and each minute;
- SDISPLAY (not described in this paper), which displays the current time (signals SDMN, SDS, SDB) of the stopwatch.

3.5 Specifying the Control

With respect to their relation with time, we can distinguish two types of actions: the atomic actions (e.g, starting, ending or suspending an activity) for which we don't care about the time they last (they are supposed real-time) and the lasting actions. For the control of these two types of actions, SIGNAL handles two kinds of control structures: events and states (represented by boolean signals). These two control structures are specified according to the fourth methodological principle to

get high-level specifications on which verification and implementation tools are more efficient.

> *4th Design Principle*: **events are issued from boolean functions, predicates on numerical values, or states; states are described by intervals of events or compositions of intervals.**

The control of the stopwatch specified according to this principle is depicted in Fig. 7.

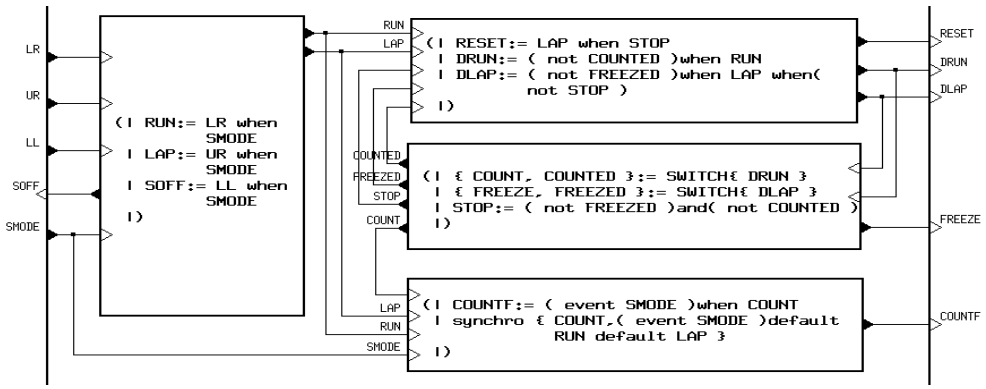


Fig. 7: The Process SCONTROL

The front-end box in Fig. 7 interprets the inputs: RUN and LAP occur when the buttons LR and UR are respectively pushed in the stopwatch mode (SMODE); the stopwatch mode is switched off (signal SOFF) on the occurrence of LL.
The two upper right boxes specify respectively events and states of the stopwatch:

- the stopwatch begins/stops counting on a RUN. The signal DRUN is used to display the state change on the screen: it holds *true* on a RUN when the stopwatch was not counting; it takes *false* when the stopwatch was counting. The state signal COUNT is switched on the occurrence of a value on DRUN;
- a LAP freezes/unfreezes the display of the current time; the current state (signal FREEZE) is switched on DLAP when the stopwatch is not stopped.
- the stopwatch is stopped (signal STOP) when it does not run and the final time (i.e., not an intermediate time) is displayed. When the stopwatch is stopped, a LAP resets (signal RESET) the current value of the stopwatch.

The lowest right box defines the counting frequency (signal COUNTF) of the stopwatch: it is the clock of SMODE when the stopwatch is counting. Note that, for the specification of the states COUNT and FREEZE, we have used a special kind of interval (process SWITCH) in which the beginning and the end are specified by the occurrences of the same signal. The generic interval process IN:=] START, STOP] is specified in the kernel of SIGNAL by:

```
(| OPEN:= START when( not IN)
 | CLOSE:= STOP when IN
 | IN:= AIN $1
 | AIN:= ( not CLOSE )default OPEN default IN
 |)
```

Further examples relying on the specification of time intervals in SIGNAL may be found in [34]. In particular, it is a good way to specify sequences of events including exceptional ones such as *interruptions* and *timeouts*, and their action on task (*suspending, resuming, stopping, ...*) in the framework of a parallel declarative language. The next version of the language will include high-level operators for defining as well as composing intervals. For instance, in this new version, the signal STOP will be directly defined as the intersection of the complementaries of COUNT and FREEZE.

3.6 Specifying Temporally-Related Computations

Programming temporally-related computations in the traditional asynchronous programming methods needs to mix effective computations with temporal instructions (e.g, synchronization instructions), the waiting periods are specified. In the SIGNAL approach, only the effective computations are specified, the clocks of the signals specifying implicitly the occurrences of the computations. This approach can be abstracted as "to act or not to be". Therefore, the specification of computational parts can be decomposed into sub-processes which highlight relevant timings of the computations when composed.

> *5th Design Principle*: **Specify computational parts into sub-processes highlighting timing issues.**

The overlap between this principle and the functional decomposition principle (2nd principle) is obvious since the notion of "functionally-coherent" may be based on timing issues, especially in real-time systems in which time is an essential feature. Applied to the process GEARS, this principle induces the decomposition in Fig. 8-a.

(a)

(b)

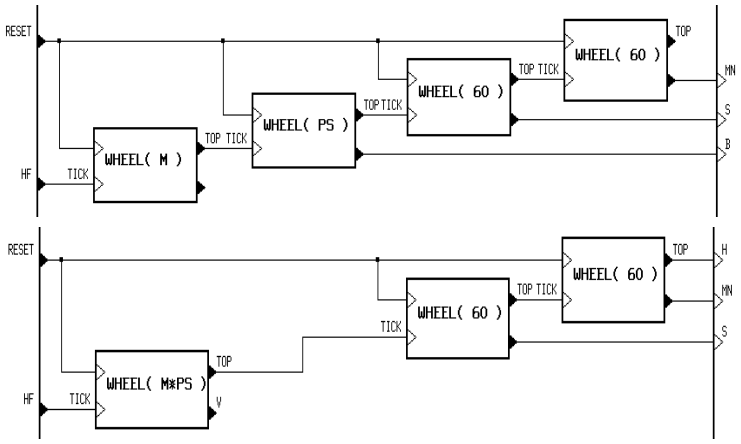


Fig. 8: (a): The Process `GEARS` (b): Illustration of the Modularity

The basic frequency of the stopwatch is given by the event-signal `HF`. The specification of `GEARS` is made up of several instantiations of the model `WHEEL` (Fig. 2, pp. 17), each of them counting the events at a local frequency `TICK` and producing a sampling of it `TOP`. From left to right, these components respectively perform (`M` and `PS` are parameters):

- every `M` ticks over the basic frequency `HF`, a `TICK` for the second component occurs. This first module samples the system time `QTZ` to get a relevant time unit (e.g, 1/100s);
- every `PS` (Per Second) occurrences of `TICK`, a second tick occurs. The parameter `PS` defines the precision of the stopwatch (`PS` is set to 100 for a 1/100s precision stopwatch). The signal `B` (basic time) carries the number modulo `PS` of events elapsed since the last `RESET`;
- every 60 occurrences of a `TICK`, a minute tick occurs. The signal `S` (second) counts the number of second ticks modulo 60;
- every 60 occurrences of a `TICK`, the counter of the minutes (signal `MN`) is reset. The hour clock (signal `TOP` of this last component) is unused for the gear of the stopwatch.

As long as the stopwatch is counting, the three units (`B`, `S`, `MN`) of the current time are incremented according to this description. They are reset to `0` when the event `RESET` occurs. The different processes `WHEEL` are connected in cascade to define the process `GEARS`: the output `TOP` (see Fig. 2) of a given process `WHEEL` defines the basic clock of the next counter, which is thus slower; each internal reset of a process `WHEEL` induces the incrementation of the following counter. For instance, the signal `MN` is present only when `S` is reset.

The modularity of the language is further illustrated in Fig. 8-b, where one of the processes `WHEEL` of `GEARS` has been removed and then the suppressed connections have been replaced. This second process can implement the gears of the watch.

3.7 Simulating SIGNAL Programs

(i) Specifying the Simulation Environment

The digital watch as described above can be simulated with a logical time without any change: it is sufficient to embed it in an environment, also described in SIGNAL, that allows its simulation. Simulating the digital watch is achieved by programming in SIGNAL the interface processes `SCANNING`, `BELL` and `DISPLAY` (Fig. 4, pp. 20). These interface processes does not need to be fully specified, they can be specialized to test particular aspects of the design. The simulation environment we designed for the digital watch focuses on its control part. A run of the digital watch in this simulation environment is depicted in Fig. 9.

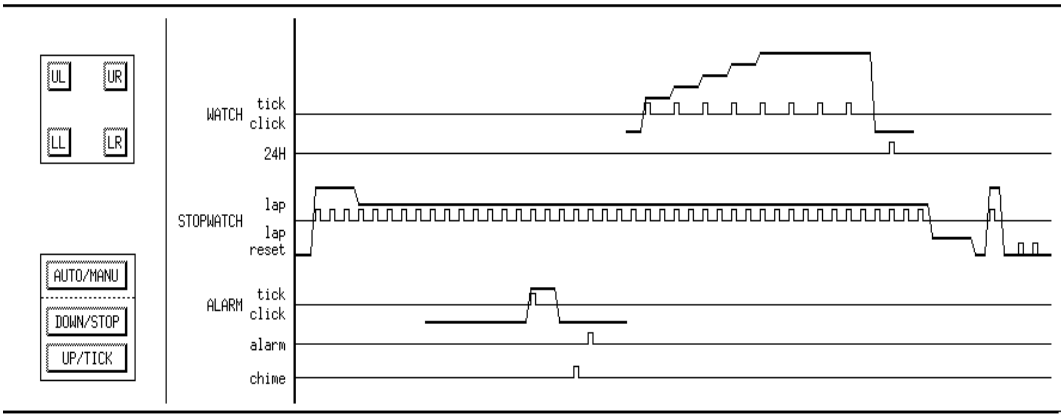


Fig. 9: Simulation of the Control Part of the Digital Watch

For the design of this simulation environment, we have slightly modified the process `SCANNING` by inserting a process `PACE_MAKER`. Like in the execution environment, the process `SCANNING` scans boolean values, corresponding to the state of the buttons of the application, set by the environment. The modification of the process `SCANNING` occurs on the basic time of the program: the time of the digital watch is given by a pace-maker controlled by three buttons AUTO/MANU, UP/TICK and DOWN/STOP. The button AUTO/MANU switches the simulation between automatic and manual modes. In the automatic mode, the time is a sampling of the system time `QTZ`. The sampling of `QTZ` is controlled by UP/TICK and DOWN/STOP which respectively speed up and slow down the simulation. In the manual mode, a time instant is generated by pushing the button UP/TICK (this allows a step by step simulation); pushing the button DOWN/STOP stops the simulation. The pace-maker makes clearly apparent the multiform nature of logical time used in SIGNAL: it is the superposition of an external time, and of events given by the user (with the button UP/TICK in manual mode).

The pace-maker is indeed a universal tool. The scanning part is quite generic too. Only the display part highly depends on both the application and the screen manager but, even this part is made up of small building blocks (e.g, the design of a clock as the physical signal) which are reusable.

As shown in Fig. 10, the pace-maker itself is embedded in its own simulation context which includes a scanner, `SCAN`, to monitor the buttons of the pace-maker, and a process `WSTOP`, which is an external process, not defined in SIGNAL, to stop the simulation.

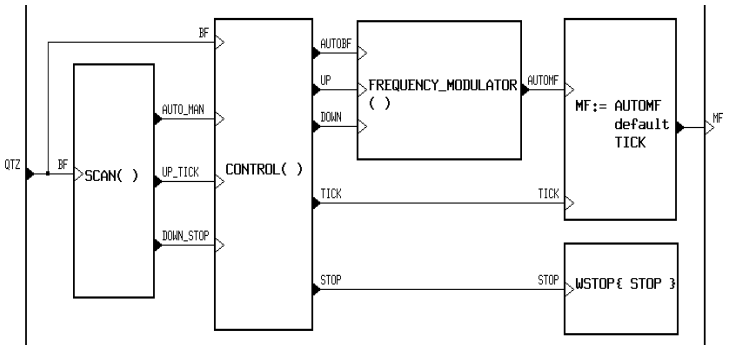


Fig. 10: The Pace-Maker with its Environment

The pace-maker receives a basic clock `BF` which is the system time `QTZ`, and, according to the mode of the pace-maker, delivers a modulated clock `MF` which either, in

automatic mode, is a sampling of QTZ depending upon the previous pushings of the buttons UP/TICK and DOWN/STOP (signal AUTOMF), or, in manual mode, is given by the button UP/TICK (signal TICK).

(ii) Introducing Implementation Issues

Further simulation of SIGNAL programs can be carried out by inserting implementation issues to the specification of an application. Thereby, it is possible to test by simulation the behavior of a specification on a particular architecture. For instance, the simulation of a distributed implementation can be specified by inserting FIFO buffers over communication links between processes; these insertions desynchronize the communication of data as it occurs through asynchronous channels of distributed architectures. This particular example emphasizes the ability to specify asynchronous behaviors within the synchronous framework of SIGNAL.

3.8 Software Reuse

(i) Black Box Abstractions

Implementing a program may require to call specific routines of the execution support (e.g. a Fast Fourier Transform Chip) or of the interface device with the environment. For instance, let us consider the specification of the process SCAN of the pace-maker. The specification of this process is depicted in Fig. 11

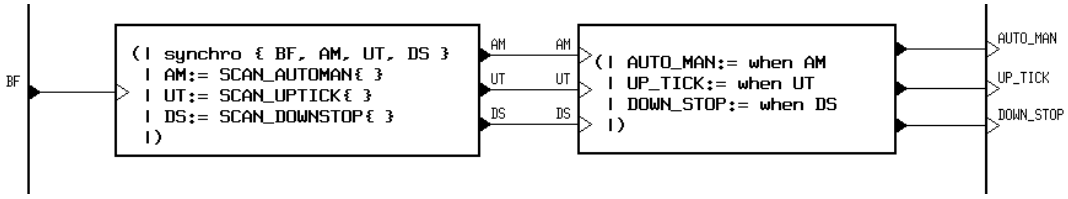


Fig. 11: The Process SCAN of the Pace-Maker

This process calls three external generative processes SCAN_UPTICK, SCAN_DOWNSTOP, SCAN_
AUTOMAN to inspect at the clock BF each one of the three buttons. For example, SCAN_UPTICK is defined by the following C function:

```
long int scan_uptick() { long int up; up = vup; vup = FALSE; return(up); }
```

with the function inter_uptick associated with the events on the corresponding button:

```
inter_uptick() { vup = TRUE; }
```

In SIGNAL, an external process is denoted by a process with no definition body. Such processes are only seen as black boxes, through their interface.

Calling specific interface routines is the first use of the notion of external processes. More generally, external processes is the medium to *reuse previously developed processes*; these processes may have been written in another language but, they can also be SIGNAL processes compiled and tested separately. Therefore, the concept of external process is the essential medium for enabling *bottom-up design methodologies. The ability of* SIGNAL *to support top-down as well as bottom-up design methodologies makes it suitable for programming large applications.*

(ii) From Black to Grey Box Abstractions

In the relation with the outside world, the ability to see and call external processes as black boxes only addresses the operational issue. The other issue, a critical one for real-time systems, is the formal verification of properties. For this purpose, we need to go from black to grey box abstractions by specifying at a certain abstract level the behavior of external processes. For the abstract specification of processes, we can still use SIGNAL since it is not just a functional language, but rather a relational one: a SIGNAL process expresses some relation, relating to clocks as well as values, between signals. Therefore, a process may describe a relation between its inputs and its outputs enforcing even some constraint on the values or on the clocks of its input signals. In particular, the SIGNAL language is able to define processes which specify *temporal safety properties.*

When an external process is used in a program, the context embedding this process has necessarily to satisfy the constraints imposed by this external process. As a typical use of this ability, a temporal safety property that should be verified by a given program can be encoded in SIGNAL as an abstraction of the environment of this program. Then, it is sufficient to compose the program with the abstraction of its environment to make sure that the desired property will be satisfied [23]: the program and the specification of the property, *both written in* SIGNAL, are transmitted and checked by the formal verification tools that the SIGNAL environment encompasses.

In general, the ability of SIGNAL to specify the abstraction of a process as a relation among its inputs and its outputs is a very important feature for software reuse since (a) it enables the user to understand at a certain level the behavior of processes which may have been developed some time ago by somebody else, and (b) it enables the SIGNAL compiler to verify if the use of an external process within a program is correct. Note that, if an external process is a SIGNAL program previously

compiled, an abstract representation of this process has already been synthetized by the compiler to enable the *separate compilation of programs.*

A final point that should be mentioned is the possibility provided by SIGNAL to *create time,* by generating instants between the instants of a given clock. This is the basic mechanism for data-dependent upsampling. It is a particular and powerful feature of the SIGNAL language that programs with upsampling can be specified —see [5] for an extensive discussion about this and the resulting expressive power.

Through the description of the digital watch, we have emphasized how the features of SIGNAL associated with a design methodology are suitable for the development of complex real-time applications satisfying software engineering requirements. With respect to the requirements of real-time systems, the definition of new languages has to be combined with the development of software environments which help the designer to devise dependable real-time implementations through automatic or semi-automatic techniques. In the next section, we present the software environment which has been designed for SIGNAL programs.

4 The SIGNAL Software Design Environment

Many software design environments include some features to cope with the timeliness requirement of real-time systems. Among these "real-time" environments, some of them focus on formal software verification and leave a huge gap between the formally proved specifications and their implementation on a particular (specific, heterogeneous, distributed, etc.) architecture. In contrast, other environments are implementation-oriented: they allow one to finely specify executions in a timely fashion, and let dependability rely exclusively on the programmer skills. Finally, a very few real-time software design environments have been designed to help the programmer in the whole design process. Even less environments offer a coherent set of tools, without gaps among the different formalisms these tools deal with.

The new real-time methodology we advocate is intended to cover all the stages of the realization of real-time systems, from their early specification up to their implementation on a particular architecture. The environment we propose offers a coherent set of tools acting on a very small number of formalisms: SIGNAL is the unique formalism used to specify the behavior and properties of the applications, and Synchronous-Flow Dependence Graphs (SFD Graphs) are the unique but polymorphic internal formalism of the SIGNAL compiler.

4.1 A Glance at the SIGNAL Design Environment

(i) Design Methodology in Three Steps

1. *Specification.*

   This new real-time methodology is based on the SIGNAL language in which real-time systems are specified. SIGNAL is intended to cover all the specification stages by successive refinements of an early specification. In this design process, top-down as well as bottom-up methodologies can be used.

   Thanks to the synchronous approach, SIGNAL specifications tend to be free from implementation-related features. This trend is reinforced by using the methodological principles presented in the previous section; these principles have been drawn to enhance the maintainability and the portability of the specifications.

2. *Verification.*

   Simulation is the first way to check properties. In comparison with the asynchronous approach, checking properties by simulation is more accurate since only deterministic behaviors are specified in SIGNAL. Despite this accuracy improvement, checking properties by simulation provides only a relative safety and, for some applications, a relative safety is clearly not sufficient. For this reason, SIGNAL's environment provides tools to prove, over architecture-independent SIGNAL specifications, properties like deadlock freedom [7], flow consistency, state exclusions and liveness [22].

3. *Implementation Inference.*

   From a specification which is correct with respect to the verifications performed, some help to infer implementations on various architectures is provided to the designer. This help is provided in three directions: (a) ensuring that the semantics of the specifications is preserved within the implementation inference process, (b) interacting with the designer to optimize the use of the target architecture features, and (c) analyzing the performance of the derived implementation to verify whether it satisfies the temporal requirements.

(ii) Overview of the SIGNAL Software Design Environment

According to SIGNAL's design methodology, the software design environment comprises three stages as depicted in Fig. 12:

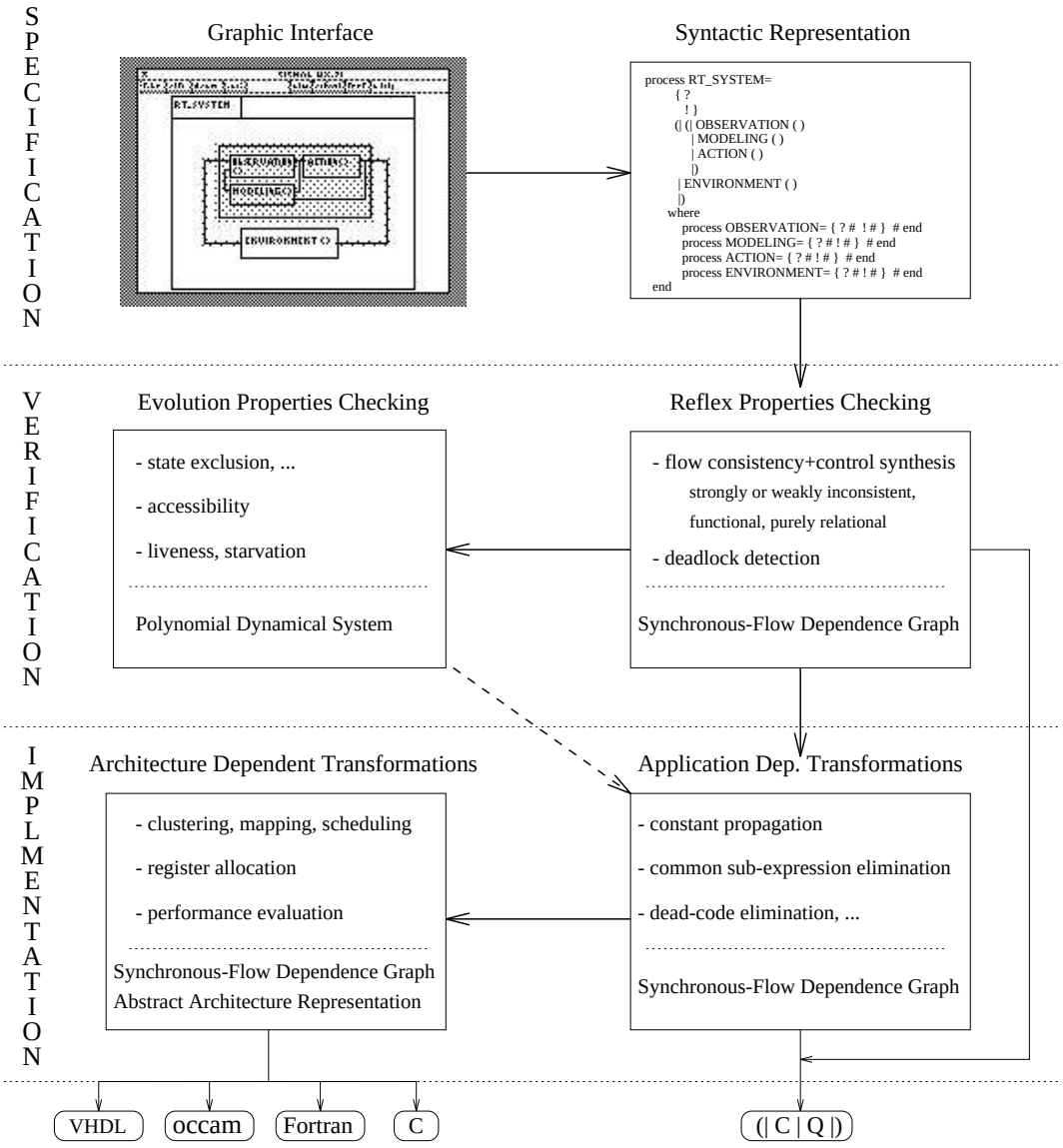


Fig. 12: The SIGNAL Software Design Environment

1. *The specification stage* is based on the syntactic form of SIGNAL programs which constitutes the input of the SIGNAL compiler. For user friendliness, a graphical interface is supplied. This interface provides the designer with a helpful way to organize his thoughts and a convenient medium to present and communicate with partners; this graphic interface has been widely used in the previous section.

2. *The verification stage* is the first layer of the compiler that SIGNAL specifications encounter.

   The synchronous approach leads applications to be specified as a discretization of their behavior in sequences of instantaneous reactions or reflexes. With respect to this decomposition, the verification stage of the SIGNAL software environment is split into two modules:

   (a) a module which focuses on (static) properties within each reflex.
   This module checks deadlock freedom and, while synthesizing the control of the flows of data, it verifies the consistency of these flows within each reflex. Four different diagnoses may result from this verification/synthesis process: strongly inconsistent, weakly inconsistent, functional and purely relational. This verification/synthesis process is sketched in section 4.2.(i) while the detection of deadlock is presented in section 4.2.(iii);

   (b) a module which checks dynamical properties.
   For checking such properties, this module has to cope with the evolution of the system over successive reflexes. This module allows one to prove dynamical properties like state exclusions, state accessibility, starvation, etc., by checking state trajectories in Polynomial Dynamical Systems [22]. The basic techniques used in this module are presented in section 4.2.(ii).

   Both modules handle the same formalism called Synchronous-Flow Dependence Graph (SFD Graph). Whereas the second module is optional, using the first module is mandatory since only deadlock free specifications with a functional control form are allowed to enter the implementation stage of the SIGNAL software design environment; the second verification module is provided to prove further properties, in particular when the relative safety induced from simulation is not sufficient.

3. *The implementation stage* is also divided into two modules to bring architecture independence, and thereby reusability, to its maximum. The first module

performs architecture independent transformations by rewriting SFD graphs; any of these transformations can be decompiled into a new SIGNAL program. The main techniques used in this module are presented in section 4.3.

The second module generates code for several architectures and provides tools to analyze the performance of the derived implementation. This module is only partially implemented in the SIGNAL compiler: the implementation of SIGNAL programs on heterogeneous distributed architectures and the analysis of their performance are enabled by means of the coupling with the SYNDEX environment [21]. The SYNDEX system performs this inference by mapping SFD graphs over a graph representation of the architecture[3].

(iii) Synchronous-Flow Dependence Graphs

The verification tools as well as the implementation techniques integrated in the SIGNAL software design environment handle a single abstract program representation called SFD Graphs. This representation is made up of:

- a *system of equations.*

  The equation system is an algebraic representation over a finite field of the *dynamical system* [6] specified by a SIGNAL program, it constitutes an abstract interpretation of this program. Depending on the information encoded in the equation system, various verifications can be performed. Over an equation system encoding relations among *clocks*, properties of the control of the flows of data within a reflex are checked, as presented in paragraph 4.2.(i). Verifying dynamical properties is achieved over equation systems encoding relations among clocks and boolean signals; such systems, called *Polynomial Dynamical Systems*, are presented in paragraph 4.2.(ii);

- a *labeled dependence graph.*

  A dependence graph, attached to the equation system over clocks, complements the abstract representation of SIGNAL programs. Paragraph 4.2.(iii), which presents the detection of deadlocks, and sub-section 4.3, which describes the inference of implementations, are dealing with such graphs.

Although different aspects of SFD Graphs are handled by the different modules, these graphs define the unique internal formalism for the compilation of the SIGNAL

[3] In fact, the graph representation of the architecture may be an hypergraph since the target architecture may include buses.

programs. This uniqueness provides the SIGNAL software design environment with great homogeneity.

4.2 Verifying Specifications

(i) Flow Consistency & Control Synthesis with Clock Calculus

The synchronous approach leads applications to be specified as sequences of reflexes; in this paragraph, we present how, while synthesizing the control of the flows of data, the consistency of these flows within a reflex is checked. This presentation is divided into three parts which respectively present the encoding of the synchronizations achieved by the operators, the analysis of this encoding by clock calculi and the different diagnoses that may result from this analysis.

◇ *Encoding the Synchronizations*

Within a reflex, a signal may carry data or not: its status may be *present* or *absent*. These two possible status of signals are encoded as follows:

$$\textit{absent} \ : \ 0 \qquad\qquad \textit{present} \ : \ 1$$

For each signal `X`, a characteristic function $\widehat{x}$ encodes its possible status within a reflex by 0 if `X` is absent and 1 if `X` is present; $\widehat{x}$ is called the *clock* of `X`. As data may be filtered through boolean values using the `when` operator, expressing the synchronization achieved by this operator needs two other data: a boolean signal is present with the value *true* or it holds the value *false*. Consequently, we introduce for any boolean signal `B` two other clocks denoted $[b]$ and $[\neg b]$ which are equal to 1 iff `B` holds respectively *true* or *false*. Between these two clocks, the following equivalences hold:

$$(1) \qquad [b] \vee [\neg b] \ = \ \widehat{b} \qquad\qquad (2) \qquad [b] \wedge [\neg b] \ = \ \widehat{0}$$

These two equivalences intuitively mean: (1) the reflexes at which `B` is present with *true* or with *false* are the reflexes at which `B` is present with a value; (2) a signal `B` can never carry *true* and *false* in the same reflex[4]. The clock $\widehat{0}$ appearing in equation (2) denotes the clock which is never present, it is the bottom element of the algebra over clocks.

With respect to this encoding, the synchronizations achieved by the operators are translated into equations over clocks. For instance, the counter `SECOND` specified

[4] as SIGNAL is a dataflow language, it satisfies the single assignment principle.

in section 2.4 (pp. 15) is encoded as:

`(| synchro{ S, TS }`	$\widehat{s} = \widehat{ts}$	(a)
`| ZS:= S $1`	$\widehat{zs} = \widehat{s}$	(b)
`| S := ( 0 when TM )default( ZS+1 )`	$\widehat{s} = \widehat{tm} \vee \widehat{zs}$	(c)
`| MAX:= ZS>=59`	$\widehat{max} = \widehat{zs} = \widehat{v59}$	(d)
`| TM:= TS when MAX`	$\widehat{tm} = \widehat{ts} \wedge [max]$	(e)
`|)`		

According to the semantics of the operators, three kinds of clock encoding are performed:

1. *monochronous operators: equivalence encoding.* Monochronous operators (the functions and the delay operator) impose the synchrony of the referred signals. This synchrony is encoded by equivalences among the associated clocks as it happens in lines (a), (b) and (d). In line (d), $\widehat{v59}$ denotes the clock of the constant signal which carries 59.

2. `default` *operator:* $\vee$*-encoding.* Merging two signals with a `default` yields a signal occurring as soon as one of these signals occurs. This union of occurrences is encoded over the $\vee$ operator between clocks; such an encoding is performed in line (c).

3. `when` *operator:* $\wedge$*-encoding.* As specified in line (e), `TM` occurs only if `TS` occurs and `MAX` holds the value *true*. Therefore, the clock $\widehat{tm}$ is present only if `TS` occurs ($\widehat{ts} = 1$) and ($\wedge$) `MAX` holds *true* ($[max] = 1$).

◇ *Analyzing Clock Equations*

A system of clock equations encodes the synchronizations that may occur within any reflex. In this equation system, we distinguish the clocks of types $[b]$ and $[\neg b]$. These clocks are called *free clocks* since they are defined with respect to boolean signals `B` which are supposed free to carry *true* or *false*. The purpose of analyzing this equation system is two-fold:

1. *Verifying the consistency of the flows of data.*

   Over the equational modeling, the flows of data are consistent when, whatever the value (zero or one) taken by a free clock, the equation system has a solution.

Intuitively, the flows of data are consistent when, whatever the status of the boolean signals B (absent, present with *true* or present with *false*) may be in a reflex, the synchronizations occur.

2. *Synthesizing the control of the flows of data.*

   This control is synthesized by defining a function computing the solutions of the equation system or, more precisely, a function computing from the free clocks the value of the other clocks of the equation system.

This verification/synthesis goes through the division of the system of clock equations by the clock equivalence. For this division, the SIGNAL compiler handles an alternative form of the clock algebra, its lattice form, which is defined as:

$$\widehat{x} \leq \widehat{y} \quad \Longleftrightarrow \quad \widehat{x} \wedge \widehat{y} = \widehat{x} \qquad (\Leftrightarrow \ \widehat{x} \vee \widehat{y} = \widehat{y})$$

From the equations $[b] \vee [\neg b] = \widehat{b}$ and $[b] \wedge [\neg b] = \widehat{0}$ among the basic clocks, we prove:

$$[b] \wedge \widehat{b} = [b] \qquad \text{and} \qquad [\neg b] \wedge \widehat{b} = [\neg b]$$

as well as their duals $[b] \vee \widehat{b} = \widehat{b}$ and $[\neg b] \vee \widehat{b} = \widehat{b}$. Thus, the orders $[b] \leq \widehat{b}$ and $[\neg b] \leq \widehat{b}$ are implicitly verified among the basic clocks. These two orders intuitively mean that a boolean signal carries a value more often than $true$ or than $false$ respectively.

Let us illustrate how this lattice form of the clock algebra is used to divide by the clock equivalence the encoding of the counter SECOND. From equations (a), (b) and (d), we deduce that $\widehat{ts} = \widehat{s} = \widehat{zs} = \widehat{max} = \widehat{v59}$. Then, equation (e) is rewritten as $\widehat{tm} = \widehat{max} \wedge [max]$ which simplifies to $\widehat{tm} = [max]$ since $[max] \leq \widehat{max}$. Finally, equation (c) is rewritten as $\widehat{max} = \widehat{tm} \vee \widehat{max}$ which is equivalent to $\widehat{tm} \leq \widehat{max}$. This latter constraint can be omitted since $\widehat{tm} = [max]$ implies that $\widehat{tm} \leq \widehat{max}$. Finally, the division by the clock equivalence of the encoding of the counter SECOND is:

$$\widehat{ts} = \widehat{s} = \widehat{zs} = \widehat{max} = \widehat{v59} \qquad \widehat{tm} = [max]$$

◇ *Resulting Diagnoses*

We distinguish two levels of flow inconsistency:

- *Strong Inconsistency.* This diagnosis occurs when the flows of data are only consistent if a boolean signal `B` never carries data. Over the divided equation system, it is denoted by the occurrence of equations such as $[b] = [\neg b]$ which have $[b] = 0$ and $[\neg b] = 0$ as unique solution.

- *Weak Inconsistency.* This diagnosis occurs when the flows of data are only consistent if a boolean signal `B` carries a particular value (which contradicts the hypothesis that `B` is free). Over the divided equation system, it is denoted by the occurrence of equations such as $\widehat{b} = [b]$ (which impose that `B` carries *true* if `B` is present) or $\widehat{b} = [\neg b]$;

The specifications having consistent flows of data are divided into two classes:

- *Functional.* This diagnosis occurs when a function computing from the free clocks the status of the other clocks has been inferred. Over the divided equation system, this occurs when all the equivalent-clock classes except one are defined in terms of conjunction-disjunction ($\wedge$, $\vee$) of free clocks. The equivalent class without a definition denotes the class of the most frequent clocks, it constitutes the activity indicator for the whole program. For the counter `SECOND`, this class is the one including $\widehat{ts}$: the counter reacts if and only if $\widehat{ts}$ occurs, i.e. only if a second occurs.

  On the one hand, this diagnosis induces *safe* implementations of the control of the flows since it denotes that these flows are *statically* consistent. On the other hand, it induces *time-predictable* implementations since a functional way of controlling these flows has been inferred. Therefore, the functional diagnosis is the first condition that specifications must satisfy before they enter the implementation stage of the compilation process;

- *Purely Relational.* This diagnosis occurs when the function synthesis has failed. Such a diagnosis would occur if line (d) of the counter is omitted: $\widehat{tm}$ is no more defined but only constrained as less frequent than $\widehat{max}$. We consider such processes as partially defined processes, further refinements are required before enabling their implementation.

An extended discussion of the verification/synthesis process and the algorithm implementing it is presented in [1].

(ii) Evolution Analysis over Polynomial Dynamical Systems

Proving dynamical properties —properties involving several successive reflexes— requires the encoding of the status of memorized signals. This may be performed by extending the abstract interpretation of SIGNAL specifications from clocks to boolean signals. The status of clocks and boolean signals within a reflex are encoded as follows:

boolean signals	clocks
true : 1	present : 1
false : −1	
absent : 0	absent : 0

Hence, a boolean signal `B` is encoded by a variable b which takes the value 0 when `B` is absent, it holds the value 1 and -1 when `B` is carrying respectively the values *true* and *false*. In this encoding, the clock $\widehat{b}$ of a signal `B` is equivalent to b^2. As this encoding extends the clock encoding, the previous clock equations are enhanced with equations among boolean values. For instance, the process

`B := A and C` is encoded by $\begin{cases} b^2 = a^2 = c^2 \\ b = a.b.(a.b - a - b - 1) \end{cases}$

Since `and` belongs to the instantaneous functions, it imposes the synchrony among the referred signals as expressed in the first equation. The second equation translates the semantics of the `and` operator at the value level. Hence, the clock encoding of the SIGNAL operators acting on boolean signals is enhanced with an equation at the value level. The dynamical feature of these equation systems, called *Polynomial Dynamical Systems*, appears on the encoding a shift register operator acting on boolean signals since it requires the memorization of values among reflexes:

`B := A $1 (init b0)` is encoded by $\begin{cases} b^2 = a^2 \\ \xi_n = a + (1 - a^2).\xi_{n-1}, \quad \xi_0 = b0 \\ b = a^2.\xi_{n-1} \end{cases}$

The first equation translates the synchrony imposed by the delay operator. The second one expresses how the state ξ_n, which memorizes the current value of `A`, is defined. The last equation specifies that `B` takes the previous value of `A` when `A` is occurring.

Hence, a SIGNAL program is encoded by a set of equations over polynomials; this encoding constitutes an extension from the clocks to the boolean signals of

the abstract interpretation of SIGNAL programs. Checking evolution properties is performed by studying the trajectory $\xi_0, \xi_1, \ldots$ of the boolean states according to the possible sequences of reflexes. The reader interested in further information is referred to [22].

(iii) Deadlock Detection with Synchronous-Flow Dependence Graphs

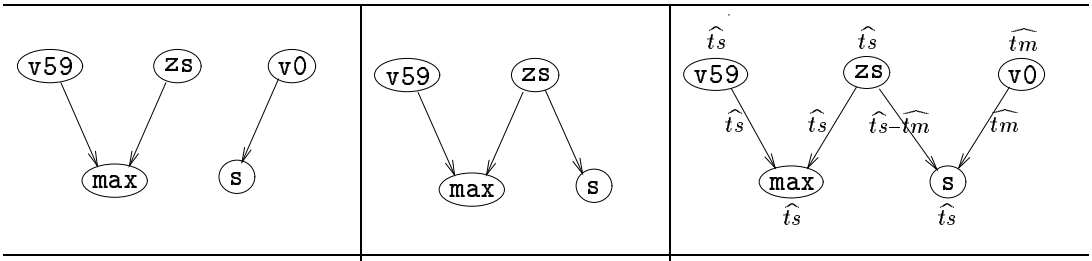
The complete abstract representation of SIGNAL programs is called Synchronous-Flow Dependence Graph (SFD Graph); it is defined by connecting a dependence graph to an equational control representation. To illustrate how this connection is performed, let us consider the counter `SECOND`. From the verification/synthesis process, we deduce:

- $\widehat{tm} = [max]$ which implies that $\widehat{tm} \leq \widehat{max}$;
- $\widehat{ts} = \widehat{s} = \widehat{zs} = \widehat{max} = \widehat{v59}$ which, consequently to $\widehat{tm} \leq \widehat{max}$, are the most frequent clocks of the counter `SECOND`: they denote the activity of the whole process.

With respect to the order relation, the possible states of $\widehat{ts}$ and $\widehat{tm}$ at which the counter `SECOND` is active are: (a) $\widehat{ts} = 1, \widehat{tm} = 1$ and (b) $\widehat{ts} = 1, \widehat{tm} = 0$. According to these two states, the different data-dependence graphs are depicted in Fig. 13-a and Fig. 13-b. All these data-dependence graphs are superimposed to define the concept of SFD graph; superimposing graphs in Fig. 13-a and Fig. 13-b induces the SFD graph drawn in Fig. 13-c. The existence conditions of the elements (nodes and vertices) of the graph are specified by clock labeling: `v0` $\xrightarrow{\widehat{tm}}$ `s` means that the value of `S` is defined by zero (`v0`) when a minute occurs ($\widehat{tm} = 1$); a similar meaning is given to the node labeling.

Deadlocks correspond to cycles in dependence graphs. Over SFD graphs, deadlocks exist iff, for any cycle of the graph, the conjunction of the clocks which label the arcs is not equal to $\widehat{0}$. In the counter `SECOND`, if `S` is reset to `W` with `W := F{S}` instead of being reset to 0, a dependence `s` $\xrightarrow{\widehat{ts}}$ `w` is added to the SFD graph in Fig. 13: a cycle `s` $\xrightarrow{\widehat{ts}}$ `w` $\xrightarrow{\widehat{tm}}$ `s` occurs. The conjunction of the labeling clocks is $\widehat{ts} \wedge \widehat{tm} = \widehat{tm}$: a deadlock exists whenever a minute occurs.

In addition to a functional control form, deadlock freedom is the second condition that a specification must verify before entering in the implementation stage of the compilation process.



(*a*) $\widehat{ts} = 1, \widehat{tm} = 1$ (*b*) $\widehat{ts} = 1, \widehat{tm} = 0$ (*c*) labeled superposition

Fig. 13: Dependence Graphs of the Process `SECOND`

4.3 Inferring Implementations

(i) Ensuring Dependability

Owing to the economical, strategic and human stakes involved in real-time computing, a particular attention must be paid to dependability.

Usually, dependability is restricted to the notion of correctness. A very common method to enhance correctness of implementations is debugging —checking behavioral properties of an implementation by an on-line monitoring of the execution. But, as monitoring affects execution performance and the behavior of real-time systems is time dependent, the debugging method is (in general) not relevant to check implementation properties of real-time systems. For this reason, correctness of real-time systems must be primarily based on the compile-time techniques which are used to infer implementations from specifications. Thus, correctness is defined as the requirement which ensures that the implementation `Impl_S` of the functional behavior of a system `S` satisfies the specification `Spec_S`. If $\wp$ denotes the set of properties on which the satisfaction condition is based, correctness can be shortened in a mathematical-like definition as:

$$\forall p \in \wp \qquad \texttt{Spec\_S} \text{ verifies } p \quad \Longrightarrow \quad \texttt{Impl\_S} \text{ verifies } p$$

The set $\wp$ includes the logical properties checked over specifications like control consistency, lack of deadlocks, state exclusions, ...

Unlike usual applications, a real-time system is a system which maintains a close relationship with its environment. For this reason, the notion of dependability for a

real-time system must not be restricted to the system in its own, it must consider its environment and be robust to its possible degradations. In other words, dependability for real-time systems must include robustness as well as correctness; this can be shortly defined as:

$$\forall p \in \wp \quad \forall \mathtt{E} \in Env \qquad (\,|\,\mathtt{Spec\_S}\,|\,\mathtt{E}\,|\,) \text{ verifies } p \quad \Longrightarrow \quad (\,|\,\mathtt{Impl\_S}\,|\,\mathtt{E}\,|\,) \text{ verifies } p \tag{3}$$

In this second definition, $\wp$ may include the previous properties as well as properties which involve the environment like (a) the timing required by the specified environment `E`; (b) the response of the system in presence of a degraded form `E'` of `E` (with $\mathtt{E}, \mathtt{E}' \in Env$) —for instance, input rate in `E'` may be a slight deviation from the rate in `E`— (c) the failure of some part of the system which imposes the use of fault-tolerance techniques.

In equation (3), checking properties over the specification is performed by applying the verification tools that have been presented in sub-section 4.2 and which act on SFD graphs. To check properties over implementations without developing new verification tools, it is sufficient to extend the concept of SFD graphs to be able to represent implementations. This problem is addressed in the next paragraph.

(ii) Inferring Fine-Grain Parallel Implementations

SFD graphs are made up of an equational control connected to a dependence graph. In paragraph 4.2.(ii), dealing with evolution properties has been performed by extending the equational part of SFD graphs from clocks to polynomials. Symmetrically, we extend the dependence graph representation from data to control to express how the equational control modeling is translated into an operational one.

In practice, the way the control is implemented can be expressed at the SFD graph level through a three-stage transformation:

1. *Node labeling transformation.* The clock labeling denotes the conditions on which the nodes depend: `s` exists iff $\widehat{ts}$=1. The clock labeling is implemented by adding dependencies from clocks to nodes such as: $\widehat{\mathtt{ts}} \xrightarrow{\widehat{ts}} \mathtt{s}$.

2. *Arc labeling transformation.*

   The clock $\widehat{tm}$ stands for the condition at which the signal `S` is reset: $\mathtt{v0} \xrightarrow{\widehat{tm}} \mathtt{s}$. If we assume that the conditioning denoted by the arc labeling is implemented in the target node, the implementation of the reset is expressed at the graph level by adding $\widehat{\mathtt{tm}} \xrightarrow{\widehat{tm}} \mathtt{s}$.

3. *Orientation of the clock equations.* The evaluation of each equivalence class is implemented as a function on boolean condition. Thus, $\widehat{tm} = [max]$ is implemented by $\texttt{max} \xrightarrow{\widehat{tm}} \widehat{\texttt{tm}}$.

By applying all these transformations to the SFD graph in Fig. 13-c, we infer the SFD graph in Fig. 14 which specifies an implementation of the counter SECOND.

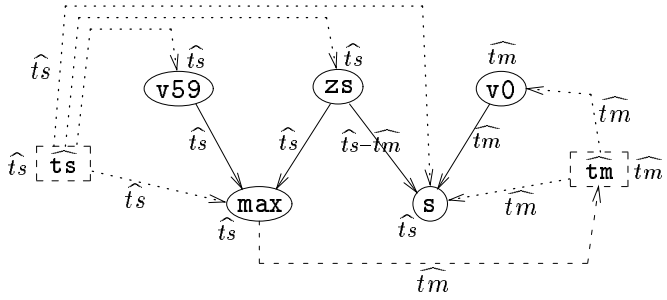


Fig. 14: An Implementation of the Process SECOND

A detailed presentation of the inference of fine-grain parallel implementations requires technical considerations which are beyond the scope of this paper —see [28, 27] for further information on this topic. From this kind of SFD graphs, the SIGNAL compiler generates sequential code (FORTRAN as well as C programs); the performance of generated programs are very near from the programs written directly in FORTRAN and C: the ratio is around 1.5 (depending on the programs); this performance can still be increased by implementing other optimizations but our present effort is mainly towards the inference of efficient distributed implementation.

(iii) Introducing Modularity with SFD Graph's Abstraction

In the previous section, we focus on equation (3) on the system S and its implementation. Another point of view on equation (3) may be to focus on the environment and to consider the symmetry between the system and the environment. In other words, if we assume that E is equal to $(|\ \texttt{S}'\ |\ \texttt{E}'\ |)$, the following equation must be verified:

$$(|\ \texttt{Spec\_S}\ |\ \texttt{Spec\_S}'\ |\ \texttt{E}'\ |) \text{ verifies } p \quad \Longrightarrow \quad (|\ \texttt{Impl\_S}\ |\ \texttt{Impl\_S}'\ |\ \texttt{E}'\ |) \text{ verifies } p$$

By enabling the composition of implementations, a first step towards the definition of the separate compilation of SIGNAL programs is performed. Completing this separate compilation process requires an abstraction tool. An abstraction is intended to

supply the sufficient information to compose processes correctly, leaving aside any superfluous features: it is the key concept for *modularity* and software re-use [20]. In programming languages, an abstracted process if often confined to an identifier and a set of input/output nodes. In some high-level languages, process abstraction may include (a) formal parameters to get more generic programs or (b) some high-order inputs like the procedure entry level in ADA [36].

More generally, the concept of abstraction is designed to assert global properties by verifying them on the composition of synthesized representations. If S and S′ are systems and Abst_S and Abst_S′ their respective abstractions, the following equation must be satisfied:

$$\forall p \in \wp \qquad \left.\begin{array}{r} \mathtt{S} \\ \mathtt{S}' \\ (\mid \mathtt{Abst\_S} \mid \mathtt{Abst\_S}' \mid) \end{array}\right\} \text{ verify } p \quad \Longrightarrow \quad (\mid \mathtt{S} \mid \mathtt{S}' \mid) \text{ verifies } p$$

With respect to the properties of deadlock freedom and flow consistency/control synthesis, the abstraction of SFD graphs involves two synthesizing mechanisms:

- *A transitive closure of the graph.*

  The synthesis, required to verify deadlock by composition, is achieved through the transitive closure of the dependence graph and its projection (sub-graph) upon the sets of input and output nodes. The transitive closure of SFD graphs is simply computed with the two following rules.

$$\text{rule of series} \qquad \mathtt{x} \xrightarrow{\widehat{c}} \mathtt{y} \xrightarrow{\widehat{d}} \mathtt{z} \quad \Rightarrow \quad \mathtt{x} \xrightarrow{\widehat{c} \wedge \widehat{d}} \mathtt{z}$$

$$\text{rule of parallel} \qquad \left.\begin{array}{l} \mathtt{x} \xrightarrow{\widehat{c}} \mathtt{y} \\ \mathtt{x} \xrightarrow{\widehat{d}} \mathtt{y} \end{array}\right\} \quad \Rightarrow \quad \mathtt{x} \xrightarrow{\widehat{c} \vee \widehat{d}} \mathtt{y}$$

- *A clock equation projection.*

  This control projection synthesizes the relations (equivalence, inclusion, exclusion) among the clocks which label the elements of the projected graph.

The counter SECOND has one input, the clock-signal TS, and two outputs, the signal S and the clock-signal TM; the abstraction of its SFD graph in Fig. 14 is presented in Fig. 15.

The concept of abstraction over SFD graphs constitutes the key point for the separate compilation of SIGNAL. Another use of this concept will be to infer efficient

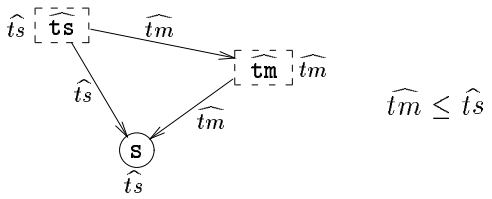


Fig. 15: Abstraction of the Implementation of the Process SECOND

distributed implementations. Although distributed implementations of SIGNAL programs are currently available through the coupling of the SIGNAL software environment with the SYNDEX system, the cooperation between these two systems needs to be improved. Currently, this coupling is based on a decompilation of SFD graphs specifying fine-grain parallel implementations. We intend to enhance this cooperation by decompiling medium-grain SFD graphs thereby improving the mapping of the application on the target architecture the grain being determined with respect to application properties —see [26] for more information.

5 Conclusion

We have presented the SIGNAL language and its software design environment for development of real-time systems. SIGNAL is a synchronous dataflow language, thereby it enables the description of behaviors in a declarative and application-oriented way. SIGNAL is not only a functional language but rather a relational one: SIGNAL programs describe behaviors as contraints between the involved signals. In this paper, we have drawn methodological principles to satisfy essential software engineering requirements for real-time programming. We have illustrated these methodological principles and the SIGNAL approach to real-time programming by the development of a complex digital watch example.

We have also given an overview of the software design environment which has been developed for SIGNAL. This environment includes formal verification tools to assert properties when the relative safety outcoming from the simulation is not sufficient. This environment includes as well tools to implement SIGNAL programs onto various architectures including distributed ones. All the tools, the verification and the implementation ones, are acting on a unique abstract program representation, namely *Synchronous-Flow Dependence Graph*. This uniqueness provides the SIGNAL software design environment with a great homogeneity.

SIGNAL is currently available under two different versions that were developed with different objectives. The INRIA H2 SIGNAL system provides the graphical interface used in this article. Sequential FORTRAN, C and VHDL code is currently produced. Although distributed implementations of SIGNAL programs are currently available by the coupling with the SYNDEX system, developments on this subject are still in progress. Moreover, tools for proving evolution properties will be integrated in a short time. The other version, the CNET-TNI V3 version called SILDEX, is commercially available from TNI Inc., Brest, France. In SILDEX, a graphical environment is provided for both program editing and on-line monitoring and supervision of the execution. A particularly important feature of SILDEX is its ability to mix SIGNAL's declarative style which the imperative style of GRAFCET [12]. In SILDEX, C, FORTRAN, or ADA code is produced. Both INRIA and SILDEX environments of SIGNAL have been experimented on significant applications in the area of signal processing and control: a speech recognition system, a radar system, a rail road crossing, were the major ones.

ACKNOWLEDGEMENTS: *The authors gratefully acknowledge Albert Benveniste, Matthew Morley and Eric Rutten for their fruitful remarks on earlier versions of this manuscript.*

References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. *Arborescent Canonical Form of Boolean Expressions.* Research Report 2290, INRIA, June 1994.

[2] T. P. Baker and O. Pazy. Real-time features for ADA 9X. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 172–180, December 1991.

[3] M. Belhadj. VHDL & SIGNAL: a cooperative approach. In *Int. Conference on Simulation and Hardware Description Languages*, pages 76–81, Society for Computer Simulation, 1994.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sep. 1991.

[5] A. Benveniste and P. Le Guernic. *A denotational theory of synchronous communicating systems.* Research Report 685, INRIA, June 1987.

[6] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[7] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[8] G. Berry. *Programming a Digital Watch in* ESTEREL *v3*. Technical Report, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1989.

[9] G. Berry. Real time programming: special purpose or general purpose languages. In G. X. Ritter, editor, *Information Processing 89*, Elsevier Science Publishers B.V., 1989.

[10] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] P. Bournai, B. Chéron, T. Gautier, B. Houssais, and P. Le Guernic. SIGNAL *manual*. Research Report 1969, INRIA, Sep. 1993.

[12] R. David and H. Alla. *Petri Nets and Grafcet: tools for modelling discrete event systems*. Prentice Hall, 1992.

[13] N. Gehani and K. Ramamrithan. Real-Time Concurrent C: a language for programming dynamic real-time systems. *The Journal of Real-Time Systems*, 3:377–405, 1991.

[14] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1321, Sep. 1991.

[16] D. Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[17] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Springer-Verlag, New York, 1985.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[19] K. M. Kavi and S. Yang. Real-time systems design methodologies: an introduction and a survey. *The journal of Systems and Software*, 85–99, Apr. 1992.

[20] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[21] C. Lavarenne, O. Segrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *European Control Conference*, pages 1684–1689, June 1991.

[22] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical systems over Galois fields and DEDS control problems. In *Proc. of the 30th IEEE conference on Decision and Control*, pages 1505–1510, IEEE Control System Society, 1992.

[23] B. Le Goff, A. Benveniste, C. Figueira, and P. Le Guernic. CAD environment for real-time control system. In *Proceedings of the 1989 American Control Conference*, pages 2666–2671, Pittsburgh, Pennsylvania, June 1989.

[24] P. Le Guernic. SIGNAL : description algébrique des flots de signaux. In *Architecture des machines et systèmes informatiques*, pages 243–252, AFCET, Hommes et techniques, 1982.

[25] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sep. 1991.

[26] O. Maffeïs and P. Le Guernic. Distributed implementation of SIGNAL: scheduling & graph clustering. In *3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 547–566, Lecture Notes in Computer Science no 863, Springer-Verlag, Sep. 1994.

[27] O. Maffeïs and P. Le Guernic. From SIGNAL to fine-grain parallel implementations. In *Int. Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, IFIP A-50, North-Holland, Aug. 1994.

[28] O. Maffeïs and P. Le Guernic. *From Synchronous-Flow Dependence Graphs to Reliable and Efficient Implementations.* Research Report, ERCIM, Jan. 1994.

[29] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[30] F. Maraninchi. The ARGOS language: graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Oct. 1991.

[31] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[32] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[33] G. M. Reed and A. W. Roscoe. Timed CSP: theory and pratice. In *Proc. of the REX Workshop on real-time: Theory in Practice*, page , 1991.

[34] E. Rutten and P. Le Guernic. Sequencing data flow tasks in SIGNAL. In *Proc. of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time System*, June 1994.

[35] W. M. Turski. Time considered irrelevant for real-time systems. *BIT*, 28:473–486, 1988.

[36] USDD. *Reference Manual for the* ADA *Programming Language.* United States, Department of Defense, 1983. ANSI:MIL-STD-1815A-1983.

[37] G. Whitrow. *Natural Philosophy of Time.* Oxford University Press, 2nd edition, 1980.

Contents

List of Figures

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399